

And Now For Something Completely Different

Programming with Python

CSE/IT 107L

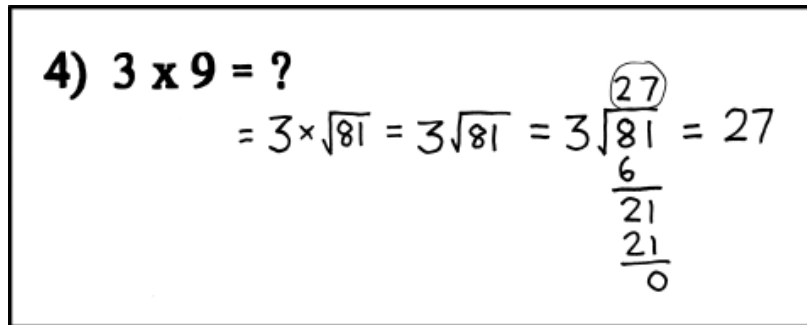
NMT Department of Computer Science and Engineering

“A beginning is the time for taking the most delicate care that the balances are correct.”

— Frank Herbert (*Dune*)

“I choose to believe what I was programmed to believe.”

— Robot Villager #2 (*Futurama*)


$$\begin{aligned} 4) \quad 3 \times 9 &= ? \\ &= 3 \times \sqrt{81} = 3 \sqrt{81} = 3 \sqrt{\frac{27}{6}} = 27 \end{aligned}$$

$\begin{array}{r} 6 \\ 21 \\ 21 \\ 0 \end{array}$

Figure 1: <http://xkcd.com/759>

Introduction

Computer programs are written using programming languages. Python is a relatively simple and frequently used programming language. It powers several popular applications, but we will use it to learn the fundamentals of writing computer programs. This lab will help you learn the basics of reading, writing and running Python programs.

You will learn how to use Python to compute arithmetic expressions, assign the results to variables, debug your programs with the help of Python’s error messages, draw pictures using the Turtle module, and run statements multiple times using for loops — the most important lesson in this lab.

Contents

1	Starting Python	1
2	Arithmetic, Variables, and Values	1
2.1	Python As A Calculator	1
2.1.1	Addition With +	1
2.1.2	Subtraction With -	1
2.1.3	Multiplication With *	2
2.1.4	Division With / and //	2
2.1.5	Modulus With %	2
2.1.6	Exponentiation With **	3
2.1.7	Negation and parentheses	3
2.2	First Error	4
2.3	Integers and Floating Point Numbers	4
2.4	Saving Values in Variables	4
2.4.1	Visualize Python Code as it Runs	6
3	Writing Python Files	6
3.1	Show Results by Using the <code>print()</code> Function	7
3.2	About Functions	8
3.2.1	Parameters are Input	8
3.2.2	Returned Values are Output	8
3.3	Read User Input Using the <code>input()</code> Function	8
3.4	Convert User Input Into Numbers by Using the <code>int()</code> and <code>float()</code> Functions	9
3.5	Debugging Programs	10
4	Repetition with <code>for</code> Loops	10
4.1	Nested For Loops	11
4.2	For Loops Dependent on User Input	12
5	Drawing Pictures With the Turtle Module	12
5.1	Drawing Functions	12
5.2	Control Drawing Speed	13
6	Sample Program	14
7	Exercises	16
	Submitting	19

1 Starting Python

To start the Python interactive shell, type the command `python3` in your terminal. It is important that you not use `python`, but `python3`. The output of this command should resemble this sample:

```
1 $ python3
2 Python 3.0.0 (default, Jun 2 2004)
3 [GCC 2.0.0 20041212] on linux
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>>
```

Note the first line of output on line 2: `Python 3.0.0`. Make sure you have *Python 3* or higher. In these labs, sample text from sessions in the Python interactive shell are surrounded in a box and have `>>>` before text that you are suppose to type. The `$ python3` and the version output are not shown. Here's an example:

```
1 >>> # Text that appears in console font within a framed box and is preceded by
2 >>> # three greater-than signs contains code and results from a Python
3 >>> # interactive shell.
4 >>> print('this is some example output')
5 this is some example output
6 >>> exit()
```

Type `exit()` then press enter, or type the key combination `Ctrl+D` to exit the Python interactive shell and go back to the terminal.

2 Arithmetic, Variables, and Values

2.1 Python As A Calculator

2.1.1 Addition With +

At the Python prompt `>>>` you can type in Python statements and immediately see the result. For instance, to add `2 + 3`, simply type `2 + 3`:

```
1 >>> 2 + 3
2 5
```

Notice that after a statement is executed, you are given a prompt to enter another statement. The `+` is called the addition operator.

2.1.2 Subtraction With -

```
1 >>> 2 - 3
2 -1
```

2.1.3 Multiplication With *

```
1 >>> 2 * 3
2 6
```

2.1.4 Division With / and //

/ divides the left hand operand by the right hand operand, while // does the division and cuts off the decimals.

```
1 >>> 2 / 3
2 0.6666666666666666
3 >>> 2 // 3
4 0
5 >>> 2 / 3.0
6 0.6666666666666666
7 >>> 2 // 3.0
8 0.0
9 >>> 2. / 3.
10 0.6666666666666666
11 >>> 2 / float(3)
12 0.6666666666666666
```

If you wish to drop the decimal portion, you must use //. If the result of a division drops decimal portion of the answer then that type of division is called integer division (since your answer will always be an integer) or floor division (since it will always round down).

This is an example of a major difference between Python 2 and Python 3. If you are using Python 2 some of these calculations will give different results.

2.1.5 Modulus With %

% divides the left hand operand by the right hand operand and gives you the remainder.

```
1 >>> 5 % 3
2 2
3 >>> 3 % 5
4 3
5 >>> 7 % 3
6 1
```

If you divide 7 by 3, 3 goes into 7 twice with a remainder of one. The modulus of two numbers can play an important role in many computations, and in later labs.

Modulus can, for example, tell you whether an integer is even or odd:

```
1 >>> 5 % 2
2 1
```



```

1 >>> 6 * 5 % 4
2 2
3 >>> 6 * (5 % 4)
4 6
5 >>> 6 * 5 % 4 - 6 ** 7 + 80 / 3 # ((6 * 5) % 4) - 67 +  $\frac{80}{3}$ 
6 -279907.3333333333

```

The # indicates the start of an inline comment. Use them to remind yourself and others about the behavior of or rationale behind the code.

2.2 First Error

What happens if you try to divide a number by zero?

```

1 >>> 100.0 / 0.0
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4     100 / arbitrary
5 ZeroDivisionError: float division by zero

```

Python raises a `ZeroDivisionError` and gives you the message `float division by zero`. In later labs, we will learn how to use code to detect some Python errors.

2.3 Integers and Floating Point Numbers

There are two types of numbers in Python, integers and floating point numbers, also called ints and floats. Integers do not have a decimal point and can represent any number without a decimal point (as long as the machine has enough memory), while floats can represent numbers with decimal points but have limited range (approximately between the enormous numbers -10^{308} and 10^{308}).

Here are some examples of integers:

0, 1, 134, -3, -10000

Floats can be written using scientific notation: `5.6E14`. The number `mEx` is interpreted as $m \cdot 10^x$.

0.0, 1.0, -2.0, 4.5, -7.41412, 5.6E14, -2.333333E-9

2.4 Saving Values in Variables

Ints and floats are examples of Python values. Python values are the *result* of computations such as `2 + 2`; they are also literal numbers like `2` or `3.1415`. Think of values as the final results of calculations. We will learn about many more types of values as the course progresses.

Variables let you store values and use them in later parts of the program. They are an indispensable feature of Python. The assignment operator (`=`) is for assigning a value to a *variable*. A variable can have any name consisting of letters, numbers, and underscores. A variable name cannot start with a number. Once a variable has been assigned, it can be used in future computations as though it were a value. Variables can be reassigned, they can also refer to their previous value. For example, to increment a variable `x` this code should assign it its previous value plus one: `x = x + 1`.

```
1 >>> x = 7
2 >>> y = 5 + x
3 >>> y
4 12
5 >>> x
6 7
7 >>> x = x + 1
8 >>> x
9 8
```

In general, you should try to give your variables descriptive names so that it is clear what they are meant to store; for example:

```
1 >>> food_price = 4.50
2 >>> drink_price = 1.00
3 >>> subtotal = food_price + drink_price
4 >>> tax = subtotal * 0.07
5 >>> total = subtotal + tax
6 >>> total
7 5.885
```

```
1 >>> arbitrary = 10.0
2 >>> 100 / arbitrary
3 10.0
4 >>> arbitrary = 0.0
5 >>> 100 / arbitrary
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8     100 / arbitrary
9 ZeroDivisionError: float division by zero
```

It is important for you to know the difference between values and variables. Values that we currently know are integers and floating point numbers like 5 or 3.1415. Variables store copies of values. Can you predict the final values of x and y in the following code?

```
1 >>> x = 4
2 >>> y = x
3 >>> x = x + 1
```

Here, x was assigned to the value 4. Then, the assignment `y = x` copied the value stored in x to y. In the third line, x was assigned to the previous value of x plus one. In the end, x contains 5 and y contains 4.

Notice that variables can be reassigned. This is because the second x in `x = x + 1` refers to the previous value. A useful trick for reading assignment statements is to read them right-to-left. For example, `x = x + 1` can be read as “the previous value of x plus one should be the new value of x.”

2.4.1 Visualize Python Code as it Runs

To see this relationship between values and variables more clearly, you may use `pythontutor.com` to visualize your Python code as it runs. Ensure you are using the Python 3 visualizer. See the following images for an example session.

Write code in Python 3.3

```

1 x = 4
2 y = x
3 x = x + 1

```

Figure 2: Editing code in the Python Tutor visualization tool. Please ensure you have select a version of Python 3 before typing or copying+pasting your code. After you finish entering Python code, you will have to press a button that says “Visualize Execution.”

Python 3.3

```

1 x = 4
2 y = x
→ 3 x = x + 1

```

[Edit code](#) | [Live programming](#)

→ line that has just executed

→ next line to execute

NEW! Click on a line of code to set a breakpoint. Then use the Forward and Back buttons to jump there.

<< First

< Back

Program terminated

Forward >

Last >>

Frames

Global frame

x	5
y	4

Figure 3: Watching code run in the Python Tutor visualization tool. Final values are presented on the right under the label “Global frame.” Use the buttons “First,” “Back,” “Forward,” and “Last” to visualize each step of your program.

3 Writing Python Files

So far, you have been using Python through the interactive shell. This lets you test ideas quickly and easily. However, it is important to know how to save your code to a file and to run it later as a program. Python code that you save to a Python file works almost identically to what you type when using the Python interpreter.

In every lab document, any code that belongs in a Python file will be formatted like this:

```

1 # Text that appears in console font within a framed box is sample Python code.
2 for, while, with, ==, in, as, if, and, or, not

```

To run this type of Python code, you must save it to a file with extension `.py`. The `py` extension indicates it is a Python file. We can run it as a program by using the command `python3 filename.py`.

How can we run the previous examples as Python files? Try to type one of the previous examples in a Python file. Open a file and name it `arithmetic.py` (or any other name that ends with `.py`). Type the following:

```
1 x = 7
2 y = 5 + x
3 x
4 y
```

This code is similar to a previous example. To run this file, use the command `python3 arithmetic.py`.

```
1 $ python3 arithmetic.py
```

Unlike most Linux commands that you've seen, this command should produce no output. We will fix that in the next section.

3.1 Show Results by Using the `print()` Function

The Python interactive shell automatically prints values, but we must use a Python function called `print()` to achieve the same result when running a Python file. To display the value of `x`, write `print(x)`. Let's fix `arithmetic.py`. Change the file so it has the following contents:

```
1 x = 7
2 y = 5 + x
3 print(y)
4 print(x)
5 x = x + 1
6 print(x)
```

```
1 $ python3 arithmetic.py
2 12
3 7
4 8
```

What if you want to print something other than numbers? We can print another type of value called a string. These are written like this: `'Hello world'`, or equivalently, `"Hello world"`. They are sequences of letters, numbers, and white-space. We'll learn more about strings in later labs. Remember that `x = '5'` and `x = 5` denote different things. Open a text editor and type the following Python code:

```
1 print("Hello, world!")
```

Save your file as `hello.py` and type the command `python3 hello.py` to run it.

```
1 $ python3 hello.py
2 Hello, world!
```

`print` is a function with the specific task of printing the value of whatever is within its parentheses. It is important to note that `print("Hello, world!")` is the same as `print('Hello World!')`. The special string `'\n'` represents a newline. This means that the text after the `\n` will start on a new line. Test this with the following example:

```
1 print('Hello, world!\nHow are you?')
```

3.2 About Functions

A function is a self-contained command that performs one specific task. Like a function in math, you can give it one or more parameters. A function may also pass a value back to the user. How many parameters are appropriate and what value is passed back depends on the specific function, though. Let's look at some examples.

3.2.1 Parameters are Input

Note that functions are used (or called) by putting its name followed by parentheses and some parameters (sometimes also called arguments) to the function. A function may have none, one, or multiple parameters (the parentheses always have to be there, though). For example:

```
1 function_name()  
2 function_name(parameter)  
3 function_name(parameter1, parameter2)
```

3.2.2 Returned Values are Output

If a function produces any output, it is *returning* its output. You can save the output of a function by assigning it:

```
1 x = function1()  
2 y = function2(parameter)  
3 z = function3(parameter1, parameter2)
```

We can infer some important lessons from this sample. Some functions, such as `function1` only produce output and accept no input. The other functions can take one or more parameters and return output. You will learn how to make your own functions in later labs. Using and creating functions is a fundamental part of writing computer programs.

Think of a function as a machine that takes parameters, processes them, and gives back the return value, which may be saved by assigning it to a variable.

3.3 Read User Input Using the `input()` Function

So far, the only person in the world that can modify how your program behaves is you, by changing the program itself. But what if we want the Python file to work more like other programs, which

accept input from their users (such as the person typing the command `python3 pythonfile.py`) and change their behavior? The function `input()` will display whatever string you write enclosed in parentheses after it, just like `print()` does; but it also pauses the program and returns whatever the user types. We can assign this to a variable and then modify our program's behavior based on what the user typed. For example, type this in a file called `input.py`:

```
1 user_input = input('Type something, then press enter: ')
2 print(user_input)
```

What will this Python file print when we run it and type “hello program! <ENTER>”?

```
1 $ python3 input.py
2 Type something, then press enter: hello program!
3 hello program!
```

As promised, the function `input()` prints `'Type something, then press enter: '`. Then the program pauses and reads user input. At this point, the variable `user_input` should contain the value `"hello program!"`. It finishes by printing the value of this variable.

3.4 Convert User Input Into Numbers by Using the `int()` and `float()` Functions

What if we want to interpret the user's input as a number instead of as a string? Let's try using `input()`'s return value as a number in the Python interactive shell:

```
1 >>> user_input = input('Write a number: ')
2 Write a number: 55
3 >>> print(user_input)
4 55
5 >>> user_input + 10
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8     user_input + 10
9 TypeError: Can't convert 'int' object to str implicitly
```

This error message lets us know that we cannot use the user's input as a number because it is a string. It must be converted into an integer or into a floating point number by using the functions `int()` or `float()`. For example, let's write a program that will read the user's input and try to convert it into an integer. Then it should multiply the input by 10 and print the result. Save it to a file called `multiplyby10.py`:

```
1 user_input = input('Please enter an integer: ')
2 user_input = int(user_input)
3 print('The result is:')
4 print(user_input * 10)
```

Try to run the program three times with inputs: “55,” “55.0,” and “not numeric.”

```
1 $ python3 multiplyby10.py
2 Please enter an integer: 55
3 The result is: 550
4
5 $ python3 multiplyby10.py
6 Please enter an integer: 55.0
7 The result is:
8 Traceback (most recent call last):
9   File "multiplyby10.py", line 2, in <module>
10     user_input = int(user_input)
11 ValueError: invalid literal for int() with base 10: '55.0'
12
13 $ python3 multiplyby10.py
14 Please enter an integer: not numeric
15 The result is:
16 Traceback (most recent call last):
17   File "multiplyby10.py", line 2, in <module>
18     user_input = int(user_input)
19 ValueError: invalid literal for int() with base 10: 'not numeric'
```

This demonstrates one successful run and two failures. If the user does not enter a string that can be converted to an integer, such as a floating point number such as `55.0` or `'not a number'`, the Python program will throw a `ValueError` with messages `invalid literal for int() with base 10`, and exit. These errors are caused by the `int()` function. We will learn how to control for bad input in later labs.

Try to make and test a new program that uses the conversion function `float()` instead of `int()`. **Try to give every program that you write invalid or extreme inputs**, or consider what would happen if a program using external information (return values from `input()`) received something unusual.

3.5 Debugging Programs

A bug is an error in computer code that causes unexpected behavior. “Operators traced an error in the 1946 Mark II computer to a moth trapped in a relay, coining the term bug. This bug was carefully removed and taped to the log book” (<http://ei.cs.vt.edu/~history/Hopper.Danis.html>.) There is no definite method of fixing or preventing unintended behavior, but modern programming languages offer many features for helping you find errors in your code.

Python will try to pinpoint which part of your code produced an error. This feedback can include the file name (`multiplyby10.py`), line number (2), and a part of the code (`user_input = int(user_input)`). Pay attention to any error messages that Python prints.

4 Repetition with `for` Loops

What if we wanted to print every number from 0 to 2? Here’s a simple, but repetitive solution:

```
1 print(0)
2 print(1)
3 print(2)
```

What if we wanted to print all numbers from 0 to 100? Or 0 to a different number, dependent on user input? We can use Python's `for` loops to repeat sections of code multiple times. For example, here is another way to print all numbers from 0 to 5:

```
1 for i in range(3):  
2     print(i)
```

Note `for i in range(N)` takes the variable `i` and assigns it the values 0, then 1, and up to $N - 1$. This approach is equivalent to this code:

```
1 i = 0  
2 print(i)  
3 i = 1  
4 print(i)  
5 i = 2  
6 print(i)
```

Now it becomes easy to print the numbers from 0 to 100; simply replace the 3 with 101.

```
1 for index in range(100):  
2     print(i)
```

Note you can use Python variables, and anything else that results in an integer value, as the range limit. For example: `for i in range(n * 3 + 5)` and `for count in range(33 * 4)`. You may refer to variables that are defined outside of the `for` loop and modify them repeatedly. This example adds all numbers from 0 to 9 then prints the result:

```
1 x = 0  
2 for i in range(10):  
3     x = x + i  
4 print(x)
```

4.1 Nested For Loops

For-loops don't have to contain simple statements, any Python code can be repeated many times; even for-loops! What do you suppose the following code sample does?

```
1 n = 3  
2 for row in range(n):  
3     for column in range(n):  
4         print(row * column)
```

It assigns 0 to `row`, then sets `column` to 0, then 1, then 2. The program prints 0 three times. Then it assigns 1 to `row`, then sets `column` to 0, then 1, then 2. Now 0, 1, then 2 are printed. The last iteration assigns 2 to `row`, and again sets `column` to 0, then 1, then 2. Finally, the program prints 0, 2, and 4.

4.2 For Loops Dependent on User Input

What if we do not know the number of iterations a for-loop must execute when we're writing the Python program? One way to do this is to use an integer that was read using `input()` and converted using `int()` as the range limit. This sample program prompts the user for a positive number, then it prints all numbers from 1 to the user's input:

```
1 user_input = input('Please enter a positive number: ')
2 user_input = int(user_input)
3 for i in range(user_input):
4     print(i)
```

This program prints the sum of all numbers from 1 to the user's input:

```
1 user_input = input('Please enter a positive number: ')
2 user_input = int(user_input)
3 x = 0
4 for i in range(user_input):
5     x = x + i
6 print(x)
```

What happens if the user's input is negative? Remember, for-loops integrate perfectly with many other Python features, can you think of any other ways to use for-loops?

5 Drawing Pictures With the Turtle Module

We will use Python's Turtle module to draw on the screen. A module can be accessed by importing it; to do this, you may type `import turtle` in Python.

5.1 Drawing Functions

Once the module is imported, you have access to a group of functions for controlling the "turtle", an arrow that moves around at your command, drawing a line where it goes.

`turtle.forward(x)` Move the turtle forward x pixels.

`turtle.backward(x)` Move the turtle backward x pixels.

`turtle.left(x)` Turn the turtle left x degrees.

`turtle.right(x)` Turn the turtle right x degrees.

`turtle.pendown()` Draw the turtle's path.

`turtle.penup()` Stop drawing.

Combining these commands will let you draw more complicated shapes. This code will draw a hexagon with side length 100. Because tracing mode is off, the turtle finishes almost instantly. Usually, turtle programs close the moment they are done drawing. The function `turtle.done()` keeps the picture open.

```
1 import turtle
2
3 turtle.tracer(False)
4 length = 100
5 for i in range(6):
6     turtle.forward(length)
7     turtle.left(60)
8 turtle.update()
9 turtle.done()
```

5.2 Control Drawing Speed

You will find it convenient to change the speed at which these drawings are made. Use the `turtle.speed('fastest')` function to change the speed of the turtle. You may also call `turtle.tracer(False)` to turn off tracing and then call `turtle.update()` when you want to show the results.

6 Sample Program

Inline comments are parts of a Python program used to convey the meaning and purpose of a section of Python code. They are not read by Python, only by programmers. Use them to explain *why* you are doing something, not to repeat the tedious details. They look like:

```
1 # This is a comment.  
2 print(2 * 2)
```

Read the following program. Make sure you understand how it works.

It uses some new functions from the turtle module:

`turtle.bgcolor()` Used to change background color. White by default.

`turtle.color()` Change color of the turtle icon.

`turtle.window_width()`, `turtle.window_height()` Get width and height of the turtle's window or drawing area.

`turtle.goto()` Place turtle at specific coordinates on the screen.

`turtle.dot(diameter, color)` Draw a dot of specified diameter and color.

```
1 import turtle  
2  
3 # Program inputs.  
4 print('Welcome to the 107L1 Radially-Precise, Pre-17th Century Planetarium!')  
5 zoom = 0.005 # input('How far to zoom in?')  
6 separation = 20 # input('How far should planets be drawn?')  
7  
8 turtle.tracer(False)  
9  
10 # Screen and turtle colors.  
11 turtle.bgcolor('black')  
12  
13 # Do not draw the turtle's path.  
14 turtle.penup()  
15  
16 # Get window size.  
17 height = turtle.window_height()  
18 width = turtle.window_width()  
19  
20 # Draw (2 * decoration) ** 2 stars.  
21 decoration = 15  
22 scale = max(height, width)  
23 for col in range(-decoration, decoration):  
24     for row in range(-decoration, decoration):  
25         # row / decoration gives is a number between -1 and 1  
26         # the `scale` variable determines the space between stars  
27         y = row / decoration * scale  
28
```



```
29     # shift every other row of stars by 50%
30     x = col + 0.5 * (row % 2)
31     x = x / decoration * scale
32
33     turtle.goto(x, y)
34
35     # make every other star's radius larger
36     turtle.dot(1 + col % 2, 'white')
37
38 # Go back to the center of the window.
39 turtle.goto(0, 0)
40
41 # Move close to the edge.
42 turtle.forward(-width / 2 * 0.9)
43
44 # Draw Mercury
45 turtle.dot(2 * 2439.7 * zoom, "gray")
46 turtle.forward((2439.7 + 6051.8) * zoom + separation)
47
48 # Draw Venus
49 turtle.dot(2 * 6051.8 * zoom, "tan")
50 turtle.forward((6051.8 + 6371.0) * zoom + separation)
51
52 # Draw Earth
53 turtle.dot(2 * 6371.0 * zoom, "blue")
54 turtle.forward((6371.0 + 3389.5) * zoom + separation)
55
56 # Draw Mars
57 turtle.dot(2 * 3389.5 * zoom, "red")
58 turtle.forward((3389.5 + 6991) * zoom + separation)
59
60 # Draw Jupiter
61 turtle.dot(2 * 6991 * zoom, "orange")
62 turtle.forward((6991 + 58232) * zoom + separation)
63
64 # Draw Saturn
65 turtle.dot(2 * 58232 * zoom, "tan3")
66 turtle.forward(58232 * zoom + separation)
67
68 turtle.done()
```

Try to run it. Can you think of any modifications? We will see a more complete and concise version in lab 4, when a new type of value — the Python list — is introduced.

7 Exercises

Read Before Starting

Make sure you understand the sample program and all of the sections about Python syntax and functions before starting these exercises.

Information on submitting the assignment is on the last page.

Exercise 7.1 (conversion.py).

Write a program that reads a floating point number using the `input()` function. Assume this number is in meters and convert it to feet, yards, and miles. Then print each of these three results.

For reference, 1 meter equals 3.281 feet, 1.094 yards, and $6.214 * 10^{-4} = 0.0006214$ miles.

Here is a sample that demonstrates the functionality of the finished program:

```
1 $ python3 conversion.py
2 Please type a measurement in meters: 1
3 The measurement in feet is:
4 3.281
5 In yards:
6 1.094
7 In miles:
8 0.0006214
```

Exercise 7.2 (mean.py).

Write a program that accepts 10 floating point numbers from the user and prints the running average after each number is entered.

Here is pseudo-code (not valid Python code) for calculating the running average:

```
1 sum = 0
2 count = 0
3 running_average = 0
4 get user_input 10 times and:
5     count = count + 1
6     sum = user_input + sum
7     running_average = sum / count
```

This is how this program should behave in case the user types “1 <ENTER> 2 <ENTER> 3 <ENTER> 4 <ENTER> 5”:

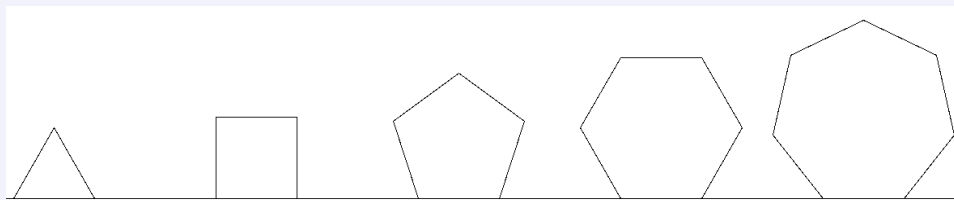
```
1 $ python3 mean.py
2 Please enter an integer: 1
3 1.0
4 Please enter an integer: 2
5 1.5
6 Please enter an integer: 3
7 2.0
8 Please enter an integer: 4
9 2.5
10 Please enter an integer: 5
11 3.0
```

And in case of 2.1, 4.7, 10.3, -4, 77:

```
1 $ python3 mean.py
2 Please enter an integer: 2.1
3 2.1
4 Please enter an integer: 4.7
5 3.4000000000000004
6 Please enter an integer: 10.3
7 5.7
8 Please enter an integer: -4
9 3.2750000000000004
```

Exercise 7.3 (polygons1.py).

Ask the user to enter a positive integer. Then consecutively draw shapes with 3, 4, 5, 6, and 7 sides where the side length of each is the user's input. For example, if the user enters 100, your program's output should resemble:

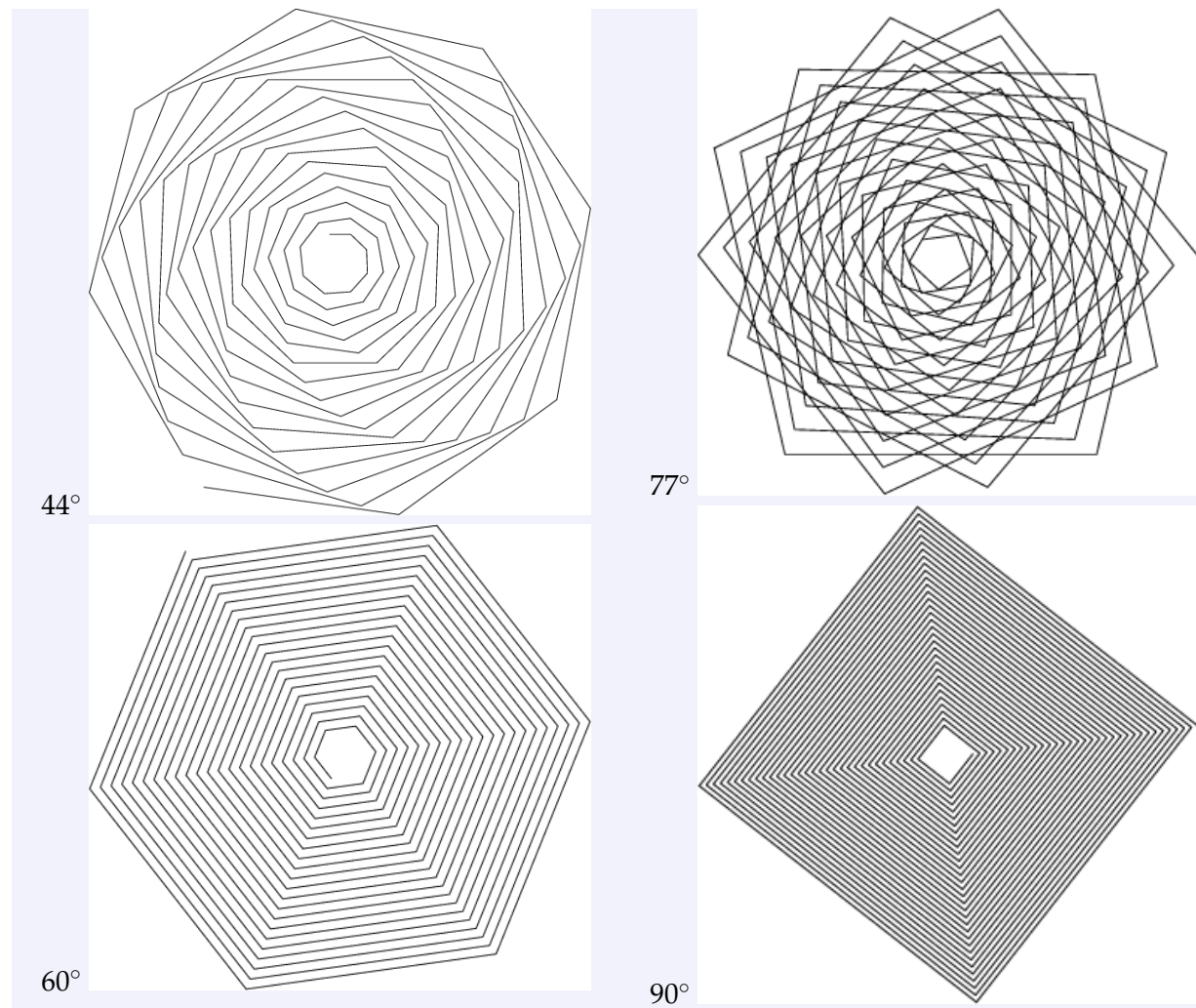


Distance between shapes does not matter, as long as they do not overlap.

Exercise 7.4 (spiral.py).

Write a script that asks the user to enter an angle. Then use turtle to draw 128 lines, each line is separated by the given angle and each line is slightly bigger than the last.

For example, if the angle is 60° then your code should draw a spiral that's made up of hexagons, as shown in the sample pictures.



Each image has 128 lines. The length and size of each component of your drawing does not have to be identical to that of the sample pictures.

Index of New Commands, Functions, and Methods

assignment, =, 4	operator +, addition, 1	turtle.goto(), 14
bug, 10	operator /, division, 2	turtle.left(x), 12
float(), 9	operator //, floor division, 2	turtle.pendown(), 12
for, 11	operator %, modulus, 2	turtle.penup(), 12
function arguments, 8	parentheses, 3	turtle.right(x), 12
function calls, 8	print(), 7, 9	turtle.speed('fastest'), 13
function parameters, 8	python3, 1	turtle.tracer(False), 13
function return value, 8	reassignment, 5	turtle.update(), 13
importing modules, 12	strings, 7	turtle.window_height(), 14
inline comments, 4	Turtle module, 12	turtle.window_width(), 14
input(), 9	turtle.backward(x), 12	unary operator -, negation, 3
int(), 9	turtle.bgcolor(), 14	ValueError, 10
newline \n, 8	turtle.color(), 14	values, 4
number, floating point, 4	turtle.done(), 13	variable, 4
number, integer, 4	turtle.dot(diameter, color), 14	ZeroDivisionError, 4
operator **, exponentiation, 3	turtle.forward(x), 12	

Review Lab 0 if you do not know how to compress your files into a .tar.gz file.

Submitting

You should submit your code as a tarball that contains all the exercise files for this lab. The submitted file should be named

`cse107_firstname_lastname_lab1.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit

7.1	Exercise (conversion.py)	16
7.2	Exercise (mean.py)	16
7.3	Exercise (polygons1.py)	17
7.4	Exercise (spiral.py)	17

Exercises start on page 16.